On Finding Multi-collisions in Random Functions

Kaushik Sarkar
Roll No: CS0804
M Tech (CS), 2st year
Year: 2009-10

M. Tech. Dissertation Thesis
Under the supervision of
Dr. Palash Sarkar

Applied Statistics Unit
Indian Statistical Institute
203, B.T. Road, Kolkata
India 700108.

Contents

Chapter 1

Introduction

Multi-collision in a function means a set of r distinct points for $r \geq 2$ in the domain of the function such that all of them map to a single point in the range of the function. In this thesis we will be mainly concerned with multi-collisions in random functions. Random functions are very important objects in modern cryptography, since MAC, hash functions and many other basic cryptographic primitives can be derived from them. In particular, 'secure' hash functions are often theoretically modeled as random functions. We know that collision-resistance is a desired property of any 'secure' hash function. 'Fast' collisions in a hash function is viewed as a certificational weakness of the construction. In the light of this discussion we can easily understand that algorithms for finding collisions in random function are of prime importance in cryptanalysis. Whereas efficient algorithms for finding two-collisions in random functions are well known and well studied, surprisingly, very little is mentioned about the more general problem of finding multi-collisions in existing literature. In this dissertation thesis we survey the existing literature on this subject as well as present our own contribution towards obtaining improved algorithms for finding multi-collisions.

Here we provide a brief outline of the thesis. In next chapter we formally introduce all the basic concepts and background necessary to understand the recent developments. In section 2.3 we start with the well known algorithms for finding two-collisions and then move on to provide the background for the main topic of this thesis, i.e. finding multi-collisions. In section 2.2 we introduce a seemingly unrelated topic – time/memory tradeoff attacks for inverting a one-way function, but later in section 2.3.6 we show how these same techniques can be used for obtaining new improved algorithms for finding three-collisions. In the last chapter we present our improvements. In section 3.1 we give the first efficient sequential algorithm for finding r-collisions for $r \geq 3$. Then we present an improved parallel algorithm for the same problem (in section 3.2) and provide the analysis in section 3.2.2. In the next section (section 3.3) we provide a new time/processor tradeoff for this problem.

Chapter 2

Part I: Preliminaries and Background

2.1 One-way Functions

Intuitively one-way function is a function that is 'easy' to compute in one direction but 'hard' to invert. We define the concept formally later in this section and explain the proper meaning of the terms like 'easy' and 'hard' used in the above informal definition. One-way functions have rather strong implications in modern cryptography. It is quite surprising to know that existence of one-way function is the necessary and sufficient assumption for achieving all of non-trivial private key cryptography. In fact there are well known constructions of pseudorandom generator, pseudorandom function and strong pseudorandom permutation given a one-way function. In a theoretical sense these combinatorial objects form all of the private key cryptography since 'secure' stream cipher are nothing but pseudorandom generators, 'secure' MAC and hash functions are pseudorandom functions and 'secure' block ciphers are essentially strong pseudorandom permutations. We should also mention that apart from private key, one-way functions also find very important place in public key cryptography also.

Now we define one-way function in a formal way. Let $f : \{0,1\}^* \to \{0,1\}^*$ a function. Consider the following experiment for any algorithm $\mathcal{A}$ and any value n for the security parameter.

Definition 1 (The inverting experiment: $\mathbf{Invert}_{\mathcal{A},f}(n)$)**.** *We define the experiment in the following 3 steps:*

1. *Choose input* $x \overset{\$}{\leftarrow} \{0,1\}^n$*. Compute* $y := f(x)$
2. *Compute* $x' := \mathcal{A}(1^n, y)$
3. *The output of the experiment is defined to* 1 *if* $f(x') = y$*, otherwise,* 0.

Now we define what we meant by the informal terms 'easy' and 'hard' in the preceding discussion.

Definition 2 (One-way function)**.** *A function* $f : \{0,1\}^* \to \{0,1\}^*$ *is one-way if the following two conditions hold:*

1. *There exists a polynomial time algorithm for computing* $f(x)$*. (this is what we described as 'easy' to compute)*
2. *For all probabilistic polynomial time algorithm* $\mathcal{A}$ *there exists a negligible function* negl*, such that*

$$\mathbf{Pr}[\mathbf{Invert}_{\mathcal{A},f}(n) = 1] \le \mathrm{negl}(n) \tag{2.1}$$

(This is what we meant by 'hard' to invert)

Quite unfortunately no function has yet been proven unconditionally one-way according to this definition. Such a proof will constitute a major breakthrough for complexity theory. But we conjecture the existence of one-way functions. These are based on some very natural problems which have been given much attention but yet to yield any probabilistic polynomial time algorithm. Here we give some examples:

- Perhaps the most famous one is the 'integer factorization' problem i.e. define $f(p, q) = pq$, then f is conjectured to be one-way where p and q are 'suitably' large primes.

- Another famous one is the 'discrete log problem': let q be a large prime factor of $p-1$ and let $\langle h \rangle$ be a subgroup of $\mathbb{Z}_p^*$ of order q. Define $f(a) = h^a \bmod p$, where $a \in \mathbb{Z}_q$. Again, f is conjectured to be a one-way function.

Taking into account the huge significance of one-way functions in cryptography, it is an important task of a cryptanalyst to see how efficiently a one-way function can be inverted (certainly this invertion cannot be done in time less than exponential, but still the question is how good we can do?). We discuss different approaches and algorithms for achieving this task in section 2.2.4. Although this invertion problem is in itself a very interesting problem that demands much attention, we mention here the algorithms and related techniques like time/memory tradeoff with another motive. In later sections (sections 2.3.6, 3.1) we show that these techniques find interesting application in the problem of finding multi-collisions in random functions.

2.1.1 Random Functions

Random function is a very important concept in cryptography since many 'secure' cryptographic schemes use random functions as a building block. Typically after proving the security of this schemes assuming a random function is used in the construction one replaces it by a pseudorandom function i.e. a function that looks like a random function to any polynomially bounded adversary. Therefore, random functions naturally capture the ideal behavior of many important cryptographic primitives. In this section we introduce the concept of random functions and briefly discuss the theoretical methods used to investigate different properties of such functions. We also list a few useful properties of the random functions.

Let us consider the set of all functions from a set X of cardinality n to a set Y of cardinality m. Let us denote this set by $\mathcal{F}_n^m$. A random function is nothing but a random variable with uniform probability distribution over $\mathcal{F}_n^m$. Another way of looking at random functions is to think of them as oracles. These oracles, when queried with a value not queried previously, return a value from the range of the function chosen uniformly at random. But these oracles are consistent, i.e. when queried with a value that has been queried already, they return the same value returned previously. We mention here that the model of random functions can be either 'exact' or 'heuristic' - in which case on the basis of simulation, we postulate that the properties of a special class of functions (e.g. quadratic function models) should be the same as the properties of the class of all functions.

We will be interested in the case where $|\mathrm{X}| = |\mathrm{Y}| = n$ (since in most of the cases, the functions that we model by random functions are length preserving functions). In general we will not view a random function as a direct represention of the mapping as a sequence of choices but instead we will consider their decomposition as a *functional graph* (basically a *random functional graph*). Mostly we will be interested in the properties of the random functions that arise from its iteration structure. Functional graph representation of a random mapping gives an excellent insight into these properties.

Let, $\varphi \in \mathcal{F}_n^n$. Consider the directed graph whose nodes are $[1, \ldots, n]$ and whose edges are ordered pairs $\langle x, \varphi(x) \rangle$, $\forall x \in [1, \ldots, n]$. Suppose, we start with u_0 and keep iterating φ, i.e. $u_1 = \varphi(u_0)$, $u_2 = \varphi(u_1), \ldots$, then before n iterations we find a value u_j, such that, it is equal to one of $u_1, u_2, \ldots, u_{j-1}$. So, in the graphical term every simple path in a functional graph connects to a cycle. The length of the path, measured in the number of the edges, is called the *tail length* of u_0 and denoted by $\lambda(u_0)$, and the length of the cycle is called - quite expectedly - *cycle length* and denoted by $\mu(u_0)$. We define the *rho length* of u_0 by $\rho(u_0) = \lambda(u_0) + \mu(u_0)$. As is clear from this discussion that functional graphs like many other objects have a very nice combinatorial structure that can be specified by a collection of other more fundamental combinatorial constructions (we give such a specification shortly). It is well known that many of the counting problems of such combinatorial objects can be directly translated into generating functions. Then we recover the asymptotic information from the singularities of such generating function using complex analysis. Here we illustrate only the first step. More details about the second step and the complete methodology can be found in [7].

As discussed above, a functional graph can be decomposed in a top-down approaching in terms of more basic combinatorial objects in the following way:

FunGraph = set(Component);
Component = cycle(Tree);
Tree = Node * set(Tree);
Node = labelled atom.

Combinatorial theory guarantees direct translation of this specification into generating function equations:

$FunGraph(z) = exp(Component(z));$
$Component(z) = log(1 - Tree(z))^{-1};$
$Tree(z) = Node(z) \times exp(Tree(z));$
$Node(z) = z.$

In the following theorem we mention some of the important properties of random graphs that can be obtained by following the above methodology. Complete proof of all the statements are given in [7]. Some these properties will be useful later in our analysis.

Theorem 1. *We can prove the following statements about a random functional graph:*

(i) The expectation of the following parameters in a random function of size n have the following asymptotic forms as $n \to \infty$,

  (a) # connected components: $\frac{1}{2}\log n$.

  (b) # nodes belonging to a cycle: $\sqrt{\pi n/2}$.

  (c) # nodes with no preimage (terminal nodes): $e^{-1}n$.

  (d) # image points: $(1 - e^{-1})n$

  (e) # k-th iterate image points: $(1 - \tau_k)n$.

  where τ_k satisfies the recurrance $\tau_0 = 0$ and $\tau_{k+1} = e^{-1+\tau_k}$

(ii) Expectation of the following parameters have the asymptotic form shown below:

  (a) Tail length (λ): $\sqrt{\pi n/8}$.

  (b) Cycle length (μ): $\sqrt{\pi n/8}$.

  (c) Rho length (ρ): $\sqrt{\pi n/2}$.

  (d) Tree size (the maximal tree containing a vertex): $n/3$.

  (e) Component size: $2n/3$.

  (f) Predecessor size (i.e. the size of the tree rooted at a vertex): $\sqrt{\pi n/8}$.

(iii) For any fixed integer r, the asymptotic form of the expectation of the following parameters are:

1. *# r-nodes (nodes with r preimages): $ne^{-1}/r!$.*
2. *# r-predecessor tree (any arbitrary tree): $nt_re^{-1}/r!$*
3. *# r-cycle tree (tree rooted on a cycle): $(\sqrt{\pi n/2}).t_re^{-r}/r!$.*
4. *# r-cycles: $1/r$.*
5. *# r-components: $c_re^{-1}/r!$*

where t_r is the number of trees having r nodes, $t_r = r^{r-1}$, and c_r is the number of connected mappings of size r.

2.2 Inversion of One-way Function

2.2.1 Basic Idea: Time/Memory Tradeoff

As we mentioned earlier that inverting a one way function is a worthy goal for the cryptanalyst, in this section we will introduce the basic ideas and concepts related to this problem. Our setting is a chosen plaintext attack scenario on a block cipher $E_k(x)$, where encryption is being done for a fixed block B. So the function $E_k(B)$ can be thought of as a function $f(k)$ which maps key space to the ciphertext space. Formally, let us assume that $f : \mathrm{X} \to \mathrm{Y}$. Clearly, if the block cipher $E_k(.)$ is 'secure' then the function f behaves like a one-way function.

One trivial approach of inverting any one way function that maps from one finite set to another finite set is to evaluate the value of the function for all the inputs and store it in a tabular form where each row consists of an ordered pair $(x, f(x))$ and then sort it on the second column. This can be termed as a 'pre-processing stage', as the calculation is done once only and before one tries to actually invert a function. Next in the 'online stage', when the value y to be inverted is given, one has to search the second column of the table prepared in the pre-processing stage and return the value which appears in the first column of the row where the given value y occurs in the second column. $\tilde{O}(N)$ time and space is required in the pre-processing stage, where $|X| = N$ and online stage requires $\tilde{O}(1)$ time and negligible memory. [1] We will later refer to this method of inversion of one way function as 'exhaustive table method'.

Another trivial approach is to avoid the pre-processing stage completely and given the value y start evaluating $f(x)$ for all $x \in X$ sequentially, until an x is found s.t. $f(x) = y$. Here no memory is required in the preprocessing stage but $\tilde{O}(N)$ time is required in online stage. We will call this method 'exhaustive search' method.

These two methods are at two extremes of time/memory trade off (TMTO) methods of one way function inversion. We will next discuss this TMTO attacks.

2.2.2 Hellman Scheme

In 1980 M. Hellman introduced the first TMTO attack which later came to be known as Hellman's method. The general idea behind the scheme is very simple. If $|X| = |Y| = N$ and we assume that the cryptanalyst is only allowed random oracle access of the one way function then under repeated evaluation of the oracle the underlaying function can be modeled as a random graph. Hellman's idea was to cover the graph by chains of certain length. In the preprocessing stage m random points on the graph is chosen as starting points for m chains, each of length t. For each starting point we repeatedly

[1] we use $\tilde{O}$ notation instead of O since we want do not want to concern ourselves with logarithmic factors as these are comparably negligible in cryptanalytic scenario. $f = \tilde{O}(g)$ if there exists c s.t. $f = O(g.log^c(g))$

evaluate the function for t times, each time on the value obtained in the previous evaluation. We store the m start and end points in a table, sorted on the end points and pass it to the online phase. In the online phase for inverting the given value y we first check whether y appears in the second column of the table. If it does not then we query the random oracle with the given value and check whether the output is present in the second column of the preprocessing table. If it is not found in the table then we again query the random oracle with the recent output value and perform the table search. We repeat this process until the value is found in the table or at most t times. If the value is found in the table then we go back to the corresponding start point and continually apply the function on it until a value x is found s.t. $f(x) = y$. However, it should be noted that finding a matching end point does not necessarily mean that the preimage of y is covered in the table since there may be collisions in the random graph. This situation is known as 'false alarm'. False alarm increases the running time of the online phase. Hellman showed that under suitably chosen parameters, the expected number of false alarms per table can bounded from above by $\frac{1}{2}$. The memory requirement in the preprocessing stage is mt and time required in the online phase to invert the given value is t.

$$\text{m chains}\left\{\underbrace{\begin{array}{l} \circ^{f()} \longrightarrow \circ^{f()} \longrightarrow \circ^{f()} \longrightarrow \cdots \rightarrow \circ^{f()} \longrightarrow \circ \\ \circ^{f()} \longrightarrow \circ^{f()} \longrightarrow \circ^{f()} \longrightarrow \cdots \rightarrow \circ^{f()} \longrightarrow \circ \\ \hfill \vdots \\ \circ^{f()} \longrightarrow \circ^{f()} \longrightarrow \circ^{f()} \longrightarrow \cdots \rightarrow \circ^{f()} \longrightarrow \circ \end{array}}_{\text{t times}}\right.$$

The probability of the success of such a scheme depends on the coverage of the underlaying random graph by the chains. Hellman showed that unless m and t is chosen s.t. $mt^2 \leq N$, the probability of collision among chains of the table becomes very high. This is known as the 'table stopping rule'. Thus by indiscriminately increasing the number of chains in the table we cannot guarantee full coverage of the graph. To increase the coverage of the random graph Hellman considered t related functions $f_1, f_2, ..., f_t$ obtained by simple output modification of the given function f, instead of a single function f. He assumed all the t different functions to be random and independent of each other. This assumption is not theoretically rigorous and has since encouraged much work in this area. Now in the preprocessing stage we make t tables, one for each of the functions f_i and in the online phase repeat the procedure described above for all the t tables. Now the memory required for the preprocessing stage is $M = mt$ and the running time $T = t^2$. These two equations together with the table stopping rule give the time/memory trade off curve $TM^2 = N^2$. If we set the parameters $m = t = N^{\frac{1}{3}}$ then we get $T = M = N^{\frac{2}{3}}$. The idea of distinguished points can be used here to reduce the number of disk accesses.

2.2.3 Rainbow Scheme

P. Oeschlin proposed a method which is known as the Rainbow scheme. Here we randomly choose mt start points and for each start point apply $f_1, f_2, ..., f_t$ sequentially and store the start point and the end point in a table and then sort the table on the end points. In the online stage first we check whether the given value y appears in the table. If this does not then we apply f_t on the given y and check whether the value appears in the table. If it does not appear then we apply f_{t-1} and f_t on y and check in the table. We proceed in this way, each time starting from one link behind. At any point if the value appears in the table we go back to the start point and finish the chain to find out the preimage of y. Here also false alarms can occur.

$$\text{mt chains}\left\{\underbrace{\begin{array}{l} \circ^{f_1()} \longrightarrow \circ^{f_2()} \longrightarrow \circ^{f_3()} \longrightarrow \ldots \rightarrow \circ^{f_t()} \longrightarrow \circ \\ \circ^{f_1()} \longrightarrow \circ^{f_2()} \longrightarrow \circ^{f_3()} \longrightarrow \ldots \rightarrow \circ^{f_t()} \longrightarrow \circ \\ \hfill \vdots \\ \circ^{f_1()} \longrightarrow \circ^{f_2()} \longrightarrow \circ^{f_3()} \longrightarrow \ldots \rightarrow \circ^{f_t()} \longrightarrow \circ \end{array}}_{\text{length t}}\right.$$

The time/memory trade off curve for Rainbow scheme is same as that of Hellman's scheme, but the running time for Rainbow is approximately half of Hellman's scheme. But as Barkan, Biham and Shamir has pointed out in their paper this gain in running time is offset by the fact that memory requirement of Hellman's scheme is half of the Rainbow scheme and that accounts for a four-fold decrease in running time for Hellman's scheme due to the form of the trade off curve $TM^2 = N^2$.

2.2.4 Parallelization of Time/Memory Tradeoff Schemes

In this section we give a brief overview of the parallel implementation of Oechslin's 'Rainbow' scheme. More details and various other implementations can be found in [8]. Here the attacker builds a circuit that can compute $f_1(f_2(\ldots(f_t(x))\ldots))$. This takes a little more time than t function evaluations. Each circuit has a constant memory. So each circuit can hold a constant number of inputs and outputs. Now the attacker builds a machine with appropriate numbers of these circuits arranged in a mesh architecture, i.e. each circuit is connected to its immediate neighbours (north, south, east and west). Most of the circuits are initialized with random starting points (these are the chains in rainbow table). But t circuits are initialized with the given value y which is to be inverted. These t circuits compute the following chains: $f_1(y),\ f_1(f_2(y)),\ldots,f_1(f_2(\ldots(f_t(x))\ldots))$. After all these chain computations are done, a sorting is applied on the output values of the circuits. We discuss a very efficient sorting algorithm suited for this situation – called Schimmler's sorting algorithm in the next paragraph. If the sorting finds collision between one of those t chains and any of the chains from the rest of the circuits then we redo the chain calculations and do the usual checks to find whether we have found a preimage of y. Here the size of the mesh (i.e. the number of the circuits) is an important parameter. [8] discusses choice of this parameter for the specific case of AES. This method can also be used for time/processor/data tradeoff.

Next we discuss the Schimmler's sorting algorithm which can sort n^2 items arranged in a mesh like network in time $\mathrm{O}(n)$ time. We will show in later sections that this sorting algorithm is very important tool for our improved parallel algorithm for finding multi-collisions.

Schimmler's sorting:
Schimmler's sorting can sort n^2 numbers in $8n - 8$ steps on two-dimensional machine, when n is a power of 2.

The machine consists of n^2 cells in an $n \times n$ mesh where each cell contains one number.and each cell is connected to its adjacent cells. There are several natural ordering of the cells in an $n \times n$ mesh. Schimmler's algorithm can sort using the left-to-right order:
$(1,1),(1,2),\ldots,(1,n),(2,1),(2,2),\ldots,(2,n),(3,1),(3,2),\ldots,(3,n),\ldots;$
the right-to-left order:
$(1,n),(1,n-1),\ldots,(1,1),(2,n),(2,n-1),\ldots,(2,1),(3,n),(3,n-1),\ldots,(3,1),\ldots$
or the snake-like order:
$(1,1),(1,2),\ldots,(1,n),(2,n),(2,n-1),\ldots,(2,1),(3,1),(3,2),\ldots,(3,n),\ldots$

Schimmler's algorithm works as follows. Recursively sort the top-left quadrant of the mesh left-to-right; the top-right quadrant left-to-right; the bottom-left quadrant of the mesh right-to-left and the bottom-right quadrant right-to-left. Sort each column independently top to bottom with odd-even transposition sort. Sort each row independently, snakelike. Sort each column independently top to bottom. Finally sort each row independently, using the desired order - left to right, right to left or snakelike.

2.2.5 Fiat-Naor's Scheme

Given the t output modification functions in Hellman's scheme, one may devise a function with a very high indegree point and rest inducing a permutation such that the Hellman's scheme fails with a very high probability. A. Fiat and M. Naor [2] gave a scheme which instead of t independent random functions as in the case of Hellman's table works with t, $t - wise$ independent functions and can invert any one way function with a constant probability. But the trade off curve $TM^3 = N^3$ is worse than Hellman's scheme.

In the preprocessing phase the scheme builds a table of the high indegree points and then in the later part when it creates chains to cover the graph, it uses this table and $t - wise$ independent functions to avoid the high indegree points. This way it can guarantee a better coverage of the graph. In the online phase it first checks whether the given point appears in the first table to decide if it is a high indegree point. If the given point is not a high indegree point then it performs a Hellman style search to locate the point in the second table.

2.3 Collision Finding in Random Functions

2.3.1 Two-collision Finding: Birthday Bound

Collision-resistance is usually cited as a desired property of a 'secure' hash function. It can be shown by using basic probability calculations that in a hash function with range size N a collision can be found with probability at least $\frac{1}{2}$ if we evaluate the hash function on $\sqrt{N}$ points in its domain. This seemingly surprising result is sometimes referred as 'The Birthday Paradox' or more technically 'The Birthday Bound'. According to this result in a group of only 22 people two persons will share a common birthday with more that 50% chance. In this section we discuss two very efficient (in terms of memory) algorithms for collision detection in hash functions. In this context we consider random functions represented as a random graph. In many cryptographic applications we are interested in the behavior of the random function under iteration. This behavior can be visualized as traversing a random path in the random graph. As mentioned earlier, usually this paths starts with a straight segment called the tail and ends in a cycle. There are many interesting algorithmic questions in this setting, like finding some point on the cycle, finding the length of the cycle or finding the cycle entry point etc. These algorithms are used in many other problems. Pollard Rho algorithm for finding small factors of a large prime uses the cycle length finding algorithm. Cycle entry points in such random paths denote a collision for the random function. Naturally collision detection algorithms for hash functions use this cycle detection algorithms. Here we discuss two cycle detection algorithms.

2.3.2 Floyd's Two Finger Algorithm

The basic intuition behind the algorithm is very simple. Suppose two persons start a race where the first person runs two times faster than the second person. After the starting point they will meet once again if and only if they are running in a circular track. This fact can be used to decide whether the track is circular or straight. Similarly in the random path we keep two pointers and move one at speed two times faster than the other until they collide. After the collision we place one of the pointers back to the starting point and move both the pointers at the same speed. The point where they collide (they will indeed collide) is the entry point of the cycle. It should be noted that this method of collision finding requires $N^{\frac{1}{2}}$ time, no better than the birthday bound but very little memory. Floyd's algorithm is suitable for finding cycles if the tail is short, but if the tail is long then the faster pointer reaches the cycle very early and traverses it many times before the slower pointer actually arrives at the cycle.

2.3.3 Nivasch's Stack Algorithm

In 2004 Nivasch [6] proposed a new algorithm which uses a single pointer and negligible amount of memory and stops immediately after recycling. The basic idea is to maintain a stack and insert new values at the time of traversing, forcing the values in the stack to be monotonically increasing. When a value is repeated at the top of the stack we stop the algorithm and find a cycle. It can be shown that the maximum size of the stack will be logarithmic in the path length. Unfortunately, though we can determine the cycle length using this algorithm, we can use it neither for Pollard's Rho nor for collision detection.

2.3.4 Multi-collision: Generalized Birthday Bound

The idea of two-collision extends to the idea of more general r-collision quite naturally. We Define an r-collision to be a set of r distinct values in the domain of a random function such that all of them map to the same value under that function. In case of arbitrary r, it is well known that finding an r-collision requires at least $(r!)^{\frac{1}{r}} \cdot N^{\frac{r-1}{r}}$ function evaluations. For small values of r we can approximate it by $\mathrm{O}(N^{\frac{r-1}{r}})$. This is in fact the generalized version of the birthday bound discussed in section 2.2. Just like two-collisions, arbitrary r-collisions also pose as security problem for hash functions and many researchers consider faster multi-collisions in a hash function as a certificational weakness.

Recent application of multi-collisions in cryptography can be found in the cryptanalysis of SHA-3 candidates Aurora-512 and JH-512. The problem of finding multi-collisions in random mapping is therefore of fundamental importance in cryptography.

The important point to note here is that the generalized birthday bound puts a lower bound on the time required in a sequential machine for finding multi-collision. It does not imply a lower bound for memory, although the most trivial algorithm will certainly require $N^{\frac{r-1}{r}}$ amount of memory - this simple algorithm involves evaluating the random function at $N^{\frac{r-1}{r}}$ distint points and storing the images in an associated array followed by a sorting step on the images to find the r-collision. But in case of two collisions, efficient algorithms with negligible memory are known (see section 2.3.2 and 2.3.3). It should be suspected that the memory requirement of the general r-collision finding algorithm can be brought down from $N^{\frac{r-1}{r}}$. In fact, [5] presents an sequential algorithm for $r = 3$ that that uses $N^{\frac{1}{3}}$ memory and $N^{\frac{2}{3}}$ time. They conjecture that the general task of finding an r-collision can be done in $N^{\frac{r-1}{r}}$ time and $N^{\frac{r-2}{r}}$ space. We present this general algorithm in section 3.1. [5] also presents a parallel algorithm for finding r-collision. We present an improved version of this algorithm in section 3.2. But for now we describe the known algorithms and algorithms presented in [5] for finding multi-collision.

2.3.5 Simple Algorithms for Three-collision

The sequential algorithms described in [5] for finding 3-collisions fits into the paradigm of time/memory tradeoff, i.e. the algorithm is divided into two phases – precomputation phase and online phase. In precomputation phase we limit the memory requirement by $\tilde{\mathrm{O}}(N^{\alpha})$ and in the online phase we bound the running time by $\tilde{\mathrm{O}}(N^{\beta})$ where $\alpha, \beta < 1$. Since we want the running time of the online phase to dominate, we will require that the running time of the precomputation phase be less than that of online phase. Ofcourse, it will be nice to have $\alpha < \beta$.

The first algorithm that we describe is named 'folklore' algorithm in [5]. It is very simple and straightforward but not as efficient as the ones to be described next. In the precomputation phase it just computes the map on N^{α} random points and stores the images in an associated array. Then it sorts the array based on the column of the image values. In the online phase it again computes the map on N^{β} random points and checks whether the image is present in the image column of the precomputed table. Whenever a hit occurs the new preimage is stored together with the old preimage in the sorted table. The algorithm succeeds in finding a three-collision if one of the N^{α} images computed in the precomputation phase is hit twice in the online phase and all the three preimages are distinct.

Now we do a heuristic analysis of the algorithm ignoring constants and logarithmic factors. The running time of the precomputaion phase is N^{α} (actually $\mathrm{O}(\alpha \cdot N^{\alpha} \log N)$ – the logarithmic factor comes due to sorting – but in this calculations it is customary to ignore the logarithmic and constant factors). Similarly, the running time of the online phase is N^{β} (actually $\mathrm{O}(\alpha \cdot N^{\beta} \log N)$ – the logarithmic factor comes due to searching operation performed on the precomputed table). Now, on average, out of the N^{β} values computed in online phase, we expect $N^{\alpha+\beta-1}$ values to hit the N^{α} values of the precomputation phase. Due to the birthday paradox, after $N^{\frac{\alpha}{2}}$ hits, we expect a double hit to occur. At that point, the algorithm succeeds if the three known preimages corresponding to the double hit are distinct, which occurs with constant probability. Therefore for the algorithm to succeed, we need:

$$\alpha + \beta - 1 \geq \frac{\alpha}{2}$$

to minimize the running time we enforce the condition:

$$\alpha + 2\beta = 2 \tag{2.2}$$

Since, we put the constraint that the running time of online phase should dominate the overall running time, we have:

$$\alpha \leq \beta \tag{2.3}$$

solving equation (2.2) and (2.3) we get $\alpha \leq \frac{2}{3}$ and $\beta \geq \frac{2}{3}$. So both the time and memory complexity of this ‘folklore’ algorithm is $\tilde{\mathrm{O}}(N^{2/3})$ – same as the trivial algorithm, though it gives a time/memory trade off according to equation (2.2). We note another point on the tradeoff curve – $\alpha = 1/2$ and $\beta = 3/4$. We will use it for comparison with the next algorithm.

The next algorithm uses a new idea to improve the time/memory tradeoff. Instead of initializing the array with N^{α} images, this algorithm initializes it with N^{α} two-collisions of the random function in the precomputation phase. To make this efficient in terms of memory use, each collision in the array is generated using a cycle finding algorithm on a (pseudo-)randomly permuted copy of the random function. Since each collision is found in time $N^{\frac{1}{2}}$ the total running time of this new precomputation phase is $N^{\frac{1}{2}+\alpha}$.

The online phase is left unchanged, we simply create N^{β} images of random points until we hit one of the known collisions. It now suffices to hit only once on a known point to succeed. As a consequence, we can replace condition (2.2) by the weaker condition:

$$\alpha + \beta = 1 \tag{2.4}$$

Since, the running time of the online phase is greater than or equal to precomputation phase, we have:

$$\alpha + \frac{1}{2} \leq \beta \tag{2.5}$$

again, solving equations (2.4) and (2.5), we get $\alpha \leq \frac{1}{4}$ and $\beta \geq \frac{3}{4}$. Comparing this with the tradeoff given by the ‘folklore’ algorithm we notice an improvement in the space complexity.

An important issue neglected in the heuristic analysis is what fraction of the candidates stored in the array in precomputation phase can be completed into a three-collision. In above analysis we tacitly assumed that all the N^α entries would yield three-collisions. This is certainly not the case. We know from section 2.1.1 and [7] that expected fraction of points with exactly k preimages is $\frac{e^{-1}}{k!}$. If P_k denotes the fraction of points with at least k preimages then we have $P_1 = 1 - e^{-1}$, $P_2 = 1 - 2e^{-1}$ and $P_3 = 1 - \frac{5}{2}e^{-1}$. The images stored in the precomputed table has at least one preimage in the 'folklore' algorithm and at least two preimages in the second algorithm. Clearly the expected fraction of entries that can be correctly completed into a three-collision is $\frac{P_3}{P_1} \approx 0.127$ for the 'folklore' algorithm and $\frac{P_3}{P_2} \approx 0.304$ for the second algorithm. To make for this loss, it suffices to increase the size of the precomputed table by a factor of 8 in the 'folklore' algorithm and a factor of 3 in the second algorithm.

In the analysis we assumed that if the algorithm returns a hit then the probability that all the three preimages of the image point will be distinct is constant. To see this, we consider only those points in the precomputed table that can be completed into a three-collision. Since in the online phase we are sampling points uniformly at random, the a *posteriori* probability of having chosen one of the two already known preimages in the second algorithm is at most $2/k$, where k is the number of distinct preimages for this point. Since $k \geq 3$, the a posteriori probability of choosing a new preimage is, at least, 1/3. Similarly, for the 'folklore' algorithm, the a posteriori probability of choosing a preimage distinct from the single originally known one is at least 2/3. To offset this loss of probability, N^β should be multiplied by a constant factor of 3.

2.3.6 Improvement Using Time/Memory Tradeoff Ideas

[5] cleverly uses Hellman's time/memory tradeoff technique for inversion of one-way functions to achieve significant improvement in the space complexity of the second algorithm. The overall structure of the algorithm remains essentially same. But Hellman's technique is used to reduce the space requirement of the N^α two-collision finding step in the precomputation phase.

The idea is to build N^α Hellman style chains with random start points, each of length N^γ. The end-point of each chain is stored together with its corresponding start-point n Hellman style table. Once the chains have been built, we sort them by end-point values as usual. Then, starting with N^α new random points, we once again compute chains of length N^γ, the difference is that we now test after each evaluation of the function whether the current value is one of the known end-points. In that case, we know that the chain we are currently computing has merged with one chain from the first step. Such merge may correspond to a collision (the false alarm situation discussed in section 2.2.2) , but certainly exception occurs when the start-point of the current chain already belongs to a precomputed chain (exactly the situation we would prefer when inverting a random function). Backtracking to the beginning of both chains, we can easily decide whether it is a collision or not and in case of collision we can construct the corresponding collision.

Since each of the two sets of chains we are constructing contains $N^{\alpha+\gamma}$ points, the expected number of collisions is $\mathrm{O}(N^{2\alpha+2\gamma-1})$. Remembering that we wish to construct N^α collisions, we need to let $\gamma = (1-\alpha)/2$. The running time necessary to compute these collisions is $N^{\alpha+\gamma} = N^{(1+\alpha)/2}$. So using Hellman's technique we can compute N^α collisions in time $N^{(1+\alpha)/2}$ instead of $N^{\frac{1}{2}+\alpha}$.

The tradeoff equation remains the same as (2.4),

$$\alpha + \beta = 1 \tag{2.6}$$

But, the constraint on the running time yields

$$\frac{1+\alpha}{2} \leq \beta \tag{2.7}$$

Solving (2.6) and (2.7), we have $\alpha \leq \frac{1}{3}$ and $\beta \geq \frac{2}{3}$, a considerable improvement in the space complexity over the 'folklore' algorithm.

2.3.7 Parallel Algorithm for Multi-collision

[5] provides a parallelizable multi-collision search algorithm. First we describe their algorithm for three-collision. The aim of the algorithm is to find three-collision using a network of low-end processor having constant amount of memory. Assume, number of processors available, $N_P \approx N^{\frac{1}{3}}$ and we aim at a running time of $\tilde{\mathrm{O}}(N^{\frac{1}{3}})$. The overlay network that we assume is a complete graph, i.e. each processor can efficiently communicate with every other processor.

The key idea behind the algorithm is *distinguished points*. By definition, a set of distinguished points is a set of points together with an efficient procedure for deciding membership. For example, the set of elements in $[0, M-1]$ can be used as a set of distinguished points since membership can be tested using a single comparison. Moreover, with this choice, the fraction of distinguished points among the whole set is simply M/N. Here, since we wish to have chains of average length $N^{1/3}$, we choose for M an integer near $N^{2/3}$. In hardware, we can simply check whether the first $1/3$ of the leading bits are zero or not to decide the membership.

The distinguished point algorithm works in two steps. During the first step, each processor starts from a random start-point s and iteratively applies the function until a distinguished point d is encountered. It then transmits a triple (s, d, L), where L is the length of the path from s to d, to the processor whose number is $d(mod N_P)$. We abort any processor if it doesn't find a distinguished point within a reasonable amount of time. Once all the paths have been computed, we start the second step. Each processor looks at the triples it now holds. If a given value of d appears three or more times, the processor recomputes the corresponding chains, using the known length information to synchronize the chains. If three of the chains merge at a common position, a 3-collision is obtained.

There are a few points to note about this algorithm:

- The interconnection network that this algorithm uses is that of a complete graph. It induces a high connection cost and it does not scale as efficiently as other interconnection structures as the the amount of communication or the number of processors connected increases. In a networked system we may use Ethernet type bus architecture to connect the processors, but due to the huge number of processors in any real application of this algorithm, very soon communication cost will dominate the runtime. The underlaying broadcast media performs very poorly under this kind of situation where the number of processors is so huge. However, if we want to implement this algorithm in a shared memory based multi-processor environment then the interconnection structure is certainly a bad one, since it scales very poorly with the number of processors. In section 3.2 we give a new version of this algorithm using a interconnection structure which is particularly well suited for VLSI implementation.

- The parallel algorithm mentioned above apparently simulates the precomputation phase of the sequential algorithm given in section 2.3.6. The total number of map evaluation is $N^{\frac{2}{3}}$ in this algorithm. But the important point to notice here is that according to the generalized birthday bound $N^{\frac{2}{3}}$ map evaluation is a necessary condition for obtaining a three-collision but it is not a sufficient condition. For example, if we compute a single chain of length $N^{\frac{2}{3}}$ starting from a random point, we will never get a three-collision (however, we may get a two collision because such long chains usually end up in a cycle). So, irrespective of the method of evaluating, $N^{\frac{2}{3}}$ map evaluation does not guarantee a three-collision. We need a proof of correctness for this algorithm. We provide a formal proof of the fact that indeed this method will yield a three-collision with constant probability in section 3.2.

The extension of the above algorithm from three-collision to r-collision is quite straightforward. Assume, number of processors, $N_P \approx N^{\frac{r-2}{r}}$. Each processor has a constant amount of memory. The integer M that defines distinguished points should be near $N^{\frac{r-1}{r}}$. Each processor first build a chain of average length $N^{\frac{1}{r}}$ (as before we abort after a reasonable number of steps), described by a triple (s, d, L). Each chain is sent to the processor whose number is $d(modN_P)$. During the second step, any processor that holds a value of d that appears in r or more triples recomputes the corresponding chains. If r chains merge at the same position, an r-collision is obtained.

We mention here that [5] stresses the use of number of processor, $N_P \approx N^{\frac{r-2}{r}}$ and length of the chain $N^{\frac{1}{r}}$. But we that believe this is just a point in the general time/processor tradeoff curve and it is as good as any other point on the curve. We argue this point in section 3.3 and provide the time/processor tradeoff curve.

Chapter 3

Part II: Our Contribution

A. Joux and S. Lucks [5] give a space efficient sequential algorithm for finding three-collisions in hash functions. We extend this algorithm for r-collision for values of $r \geq 3$. We describe the algorithm and give a heuristic analysis in sections 3.1. [5] also presents a parallel algorithm for finding r-collisions. We give an improved version of that algorithm using a better interconnection structure in section 3.2. Both our algorithm and the parallel algorithm presented in [5] make an implicit assumption. We formally prove this assumption and present a coherent theory in section 3.2.2. Also we provide a new time/processor tradeoff for finding multi-collisions of hash function in section 3.3.

3.1 General Sequential Algorithm for Multi-collision

Just like three-collisions, in the problem of finding r-collision, for values of r greater than 3, one has a number of alternative options of organizing the algorithm in precomputation and online phase. To make the discussion concrete, let us take the value r as 4. Now in case of four-collisions, there are five alternative ways of dividing the algorithm in two parts:

1. In the first phase prepare a list of image points. In the second phase take random points and compute the map until 3 points hit one of the precomputed images in the first phase.
2. The first phase is same as the previous one. But in the second phase prepare a list of three-collisions and seek one hit between the two lists.
3. In the first phase compute a list of two collisions. In the second phase also compute a list of fresh two-collision and seek a hit between the items of the two lists.
4. The first phase is same as the previous one. But, in the second phase compute the map on randomly chosen points until two of the images hit one of the precomputed two-collisions.
5. First phase consists of computing a list of three-collisions. In the second phase we evaluate the map on randomly chosen points until one image hit one of the precomputed three-collisions.

Clearly, number of such alternative options increase with the value of r. In general, it may appear that dividing the precomputation and online phase in a balanced way (i.e. $\lceil \frac{r}{2} \rceil$-collision in first phase and $\lfloor \frac{r}{2} \rfloor$-collision in the second phase) may give the optimum efficiency. However, this intuition is not correct as we show in the next section.

3.1.1 Description and Analysis

First, we describe the algorithm for $r = 4$. For four-collision, in the precomputation phase we compute N^{α} two-collisions using the Hellman's time/memory tradeoff technique. As discussed in section 2.3.6, this can be done in time $N^{\frac{1+\alpha}{2}}$ using space N^{α}. We make a table where each image is stored with its two preimages and sort the table on the image values. In the online phase we take N^{β} randomly chosen points and evaluate the map on them. After each map evaluation we check whether this image is already present in the precomputed image table. Following the line of analysis presented in [5], we can expect $N^{\alpha+\beta-1}$ of the new images to hit the precomputed table. Now if this number of hit is at least as big as $N^{\frac{\alpha}{2}}$, then by birthday bound we can expect two of the new values to hit one of the precomputed image values. If all the preimages are distinct, the probability of which is again constant, then will find a four-collision. Therefore, we have

$$\alpha + \beta - 1 \geq \frac{\alpha}{2}$$

So to minimize the running time we have the following equation:

$$\alpha + 2\beta = 2 \tag{3.1}$$

And the running time constraint gives:

$$\beta \geq \frac{1+\alpha}{2} \tag{3.2}$$

Solving (3.1) and (3.2) gives $\alpha \leq \frac{1}{2}$ and $\beta \geq \frac{3}{4}$.

For general r-collision, the scheme is same. In the precomputation phase we compute N^{α} two-collisions in time $N^{\frac{1+\alpha}{2}}$ and space N^{α}. We store the two-collisions together with their two preimages in a table, sorted on the image values. Next in the online phase, we compute the map on N^{β} randomly chosen points. We want $(r-2)$ of them to hit the same value in the precomputed two-collision table. If all the r preimages are distinct then we will have an r-collision. Since, the probability of the event that all the peimages are distinct is constant, we can ignore its effect for the time being and proceed with the analysis.

Clearly the number of hits between the two lists will be $N^{\alpha+\beta-1}$. To make $(r-2)$ of this values to hit a particular value of the precomputed table, by generalized birthday bound, we have:

$$N^{\alpha+\beta-1} \geq N^{\alpha \cdot \frac{r-3}{r-2}}$$

To minimize the running time we take the equality and after simplifying we get:

$$\alpha + (r-2)\beta = r - 2 \tag{3.3}$$

The usual constraint on running time gives:

$$\beta \geq \frac{1+\alpha}{2} \tag{3.4}$$

Now, solving equations (3.3) and (3.4), we have $\alpha \leq \frac{r-2}{r}$ and $\beta \geq \frac{r-1}{r}$. That means that we can compute r-collisions in time roughly $N^{\frac{r-1}{r}}$ using space roughly $N^{\frac{r-2}{r}}$. This is a considerable improvement over the naive algorithm in terms of space efficiency.

3.2 A New Improved Parallel Algorithm for Multi-collision

In this Section we will describe an improvement of the parallel algorithm due to A. Joux and S. Lucks [5] described in section 2.3.7. We will use techniques developed in [8]. Particularly we will use the mesh/grid interconnection structure for interconnecting the processors. This is better interconnection structure than the complete graph structure proposed in [5] in terms of interconnection cost as well as efficient VLSI implementation. Our algorithm requires same number of processors as the algorithm in [5] and asymptotic running time of our algorithm is also same as their algorithm.

3.2.1 Description

First we describe our algorithm for finding three-collision. An important observation is that the message passing step where each processor sends its (s, d, L) triple to the processor numbered $d(mod N_P)$ in the algorithm given in [5] is equivalent to sorting the triples on the d values. Our algorithm builds around this idea.

Suppose, $N_P \approx N^{\frac{1}{3}}$ processors with constant memory are arranged in a two-dimensional mesh structure of size $N^{\frac{1}{6}} \times N^{\frac{1}{6}}$. Each processor is connected to its immediate neighbours only (i.e. north, south, east and west neighbours). We take our M to be $N^{\frac{2}{3}}$, such that the average length of the chains be $N^{\frac{1}{3}}$. Each processor starts from a random start-point s and iteratively applies the function until a distinguished point d is encountered. Each processor makes a triple (s, d, L), where L is the length of the path from s to d. Then we apply Schimmler's sorting algorithm (or any other efficient parallel sorting algorithm) on the output distinguished points d. One point to note here is that though we sort on d, during the sorting we actually exchange the whole triple (s, d, L) between the processors. If there are three or more processors with same terminating distinguished points (i.e. merging between two or more chains), then this sorting step reveals them. Moreover, the sorting arranges these triples with same d value along a single row (or a column, if sorting is done in the column major way). So, now each processor checks the value of d of its east and west neighbour (assuming row-major sorting) and if the value is same as its own d value then it borrows their tuple and recomputes the corresponding chains, using the known length information to synchronize the chains. If three of the chains merge at a common position, a three-collision is obtained.

This algorithm is conceptually same as the first algorithm except that it replaces the message passing step of the first algorithm by Schimmler's sorting step. Consequently, we can use the more efficient mesh structure than the complete graph structure used in the first algorithm. Clearly, this algorithm uses same number of processors as the first one. But here we incur extra cost in terms of running time for the Schimmler's sorting step. But, as discussed in section 2.2.4 this takes only $8 \cdot N^{\frac{1}{6}}$ time. So the chain computation time $\mathrm{O}(N^{\frac{1}{3}})$ dominates. Hence the asymptotic time complexity of our algorithm is same as the first algorithm.

The structure of the general version of this algorithm for finding r-collisions is essentially same. It also involves chain computation and sorting steps followed a chain recomputation step to see whether r or more chains merge at the same point to yield an r-collision. We discuss this algorithm in more details with the choice of the appropriate parameter values (like number of processors and average chain length) after we describe the time/processor tradeoff in section 3.3.

3.2.2 Analysis

We have now seen two parallel algorithms for finding three-collisions – the first one due to [5] described in section 2.3.7 and our new algorithm presented in the previous section. Both the algorithms make the implicit assumption that computing $N^{\frac{1}{3}}$ chains of length $N^{\frac{1}{3}}$ with random starting-points will yield at least one three-collision (i.e. at least three chains will merge at a common point). We give a formal proof of this fact in this section.

In the following analysis we make the following two assumptions:

- *A single chain contains all distinct points.* During the computation of a chain repetition of any value implies that the chain has entered a cycle. Since we are using *distinguished point* method for computing the chains, repetition of any value implies that this chain will not encounter a distinguished point. So the chains with repeated values will be discarded.

- *Chain lengths are equal.* Though it will not be the case in DP method, the probability that the chain length of a particular chain is far away from the average chain length will be too small to affect our analysis.

Suppose, we choose m random points and starting with each of them we calculate chains of length t. We are interested in calculating the expected number of r-collisions for $r \geq 3$. Here we make an important observation – the i-th chain can lead to at most one r-collision for all values of $r \leq i$, since once a collision occurs the new chain merges with an old chain.

Let, X_r be a random variable that denotes the number of r-collisions under this setting. We define the following useful events:
A_i^r : the i-th chain will lead to an r-collision. $r \geq 3$

C_i^r : After the computation of the i-th chain we will have at least one r-collision where $r \geq 2$. Similarly, $\bar{C}_i^r$ denotes the probability that upto the completion of computation of the i-th chain no r-collision occurs.

$\mathrm{B}_{i,j}^r$: $x_{i,j}$ i.e. the j-th value of the i-th chain will lead to an r-collision. $r \geq 3$

Let, $\mathrm{I}(\mathrm{A}_i^r)$ denote the indicator random variable of the event A_i^r. Then $\forall r \geq 2$ we have:

$$\mathrm{X}_r = \sum_{i=1}^{m} \mathrm{I}(\mathrm{A}_i^r)$$

Therefore, we have, due to linearity of expectation:

$$\begin{aligned}\mathbf{E}(\mathrm{X}_r) &= \sum_{i=1}^{m} \mathbf{E}(\mathrm{I}(\mathrm{A}_i^r)) \\ &= \sum_{i=1}^{m} \mathbf{Pr}[\mathrm{A}_i^r] \qquad (3.5)\end{aligned}$$

Now, $\mathbf{Pr}[\mathrm{A}_i^r] = 0,\ \forall i < r$. Similarly $\forall i \geq r$, we have the following relation:

$$\mathbf{Pr}[\mathrm{A}_i^r] = 1 - \mathbf{Pr}[\wedge_{j=1}^{t} \bar{\mathrm{B}}_{i,j}^r] \qquad (3.6)$$

Using the chain rule of compound probability we get:

$$\mathbf{Pr}[\wedge_{j=1}^{t} \bar{\mathrm{B}}_{i,j}^r] = \prod_{j=1}^{t} \mathbf{Pr}[\bar{\mathrm{B}}_{i,j}^r | \bar{\mathrm{B}}_{i,1}^r, \ldots, \bar{\mathrm{B}}_{i,j-1}^r] \qquad (3.7)$$

Let us denote the conditional event $\bar{\mathrm{B}}_{i,j}^r | \bar{\mathrm{B}}_{i,1}^r, \ldots, \bar{\mathrm{B}}_{i,j-1}^r$ by $\mathrm{D}_{i,j}^r$. Therefore $\mathrm{D}_{i,j}^r$ denotes the event that given none of $x_{i,1}, \ldots, x_{i,j-1}$ has led to an r-collision, $x_{i,j}$ will also not lead to an r-collision.

$$\begin{aligned}\mathbf{Pr}[\bar{\mathrm{B}}_{i,j}^r | \bar{\mathrm{B}}_{i,1}^r, \ldots, \bar{\mathrm{B}}_{i,j-1}^r] &= \mathbf{Pr}[\mathrm{D}_{i,j}^r] \\ &= \mathbf{Pr}[\mathrm{D}_{i,j}^r \wedge \bar{\mathrm{C}}_{i-1}^{r-1}] + \mathbf{Pr}[\mathrm{D}_{i,j}^r \wedge \mathrm{C}_{i-1}^{r-1}] \qquad (3.8) \\ &= \mathbf{Pr}[\bar{\mathrm{C}}_{i-1}^{r-1}] \cdot \mathbf{Pr}[\mathrm{D}_{i,j}^r | \bar{\mathrm{C}}_{i-1}^{r-1}] + \mathbf{Pr}[\mathrm{C}_{i-1}^{r-1}] \cdot \mathbf{Pr}[\mathrm{D}_{i,j}^r | \mathrm{C}_{r-1}^{i-1}] \\ &\leq \mathbf{Pr}[\bar{\mathrm{C}}_{i-1}^{r-1}] + (1 - \mathbf{Pr}[\bar{\mathrm{C}}_{i-1}^{r-1}])(1 - \frac{1}{N}) \\ &= 1 - \frac{1}{N} + \frac{\mathbf{Pr}[\bar{\mathrm{C}}_{i-1}^{r-1}]}{N} \qquad (3.9)\end{aligned}$$

We make an important observation here – the step (3.8) cannot be derived for $r = 2$. So the inequality (3.9) is valid only for $r \geq 3$. Clearly, the probability $\mathbf{Pr}[\mathrm{D}_{i,j}^r | \bar{\mathrm{C}}_{i-1}^{r-1}] = 1$, since, if there are no $(r-1)$-collision upto the $(i-1)$-st chain, then $x_{i,j}$ cannot yield an r-collision. At any stage to get an r-collision we need an existing $(r-1)$-collision. $\mathbf{Pr}[\mathrm{D}^r_{i,j}|\mathrm{C}^{i-1}_{r-1}]$, i.e. given that upto $(i-1)$-st chain there is at least one $(r-1)$-collision and the i-th chain has not yet led to an r-collision, the probability that $x_{i,j}$ will not lead to an r-collision can be upper bounded by $(1-\frac{1}{N})$.

Next we prove a recurrence relation for the quantity $\mathbf{Pr}[\bar{\mathrm{C}}^r_i]$.

Lemma 1. $\mathbf{Pr}[\bar{\mathrm{C}}^r_i] \le \prod_{k=r}^{i}(1-\frac{1}{N}+\frac{\mathbf{Pr}[\bar{\mathrm{C}}^{r-1}_{i-1}]}{N})^t$*, for all* $r \ge 3$

Proof: The proof is straightforward. From the definition, we get the following relation:

$$\mathbf{Pr}[\bar{\mathrm{C}}^r_i] = \mathbf{Pr}[\wedge^i_{k=1}\bar{\mathrm{A}}^r_k] \tag{3.10}$$

Now, using the chain rule of compound probability we get:

$$\begin{aligned}
\mathbf{Pr}[\bar{\mathrm{C}}^r_i] &= \mathbf{Pr}[\wedge^i_{k=1}\bar{\mathrm{A}}^r_k] \\
&= \prod_{k=1}^{i}\mathbf{Pr}[\bar{\mathrm{A}}^r_k|\bar{\mathrm{A}}^r_1,\dots,\bar{\mathrm{A}}^r_{k-1}] \\
&= \prod_{k=1}^{i}\mathbf{Pr}[\wedge^t_{j=1}\bar{\mathrm{B}}^r_{k,j}|\bar{\mathrm{A}}^r_1,\dots,\bar{\mathrm{A}}^r_{k-1}] \\
&= \prod_{k=1}^{i}\prod_{j=1}^{t}\mathbf{Pr}[\bar{\mathrm{B}}^r_{k,j}|\bar{\mathrm{B}}^r_{k,1},\dots,\bar{\mathrm{B}}^r_{j-1},\bar{\mathrm{A}}^r_1,\dots,\bar{\mathrm{A}}^r_{k-1}] \\
&= \prod_{k=1}^{i}\prod_{j=1}^{t}\mathbf{Pr}[\mathrm{E}^r_{k,j}] \\
&\le \prod_{k=1}^{i}\prod_{j=1}^{t}(1-\frac{1}{N}+\frac{\mathbf{Pr}[\bar{\mathrm{C}}^{r-1}_{k-1}]}{N}) && (3.11) \\
&= \prod_{k=1}^{i}(1-\frac{1}{N}+\frac{\mathbf{Pr}[\bar{\mathrm{C}}^{r-1}_{k-1}]}{N})^t && (3.12) \\
&= \prod_{k=r}^{i}(1-\frac{1}{N}+\frac{\mathbf{Pr}[\bar{\mathrm{C}}^{r-1}_{k-1}]}{N})^t && (3.13)
\end{aligned}$$

The event $\bar{\mathrm{A}}^r_k$ is equivalent to the event $[\wedge^t_{j=1}\bar{\mathrm{B}}^r_{k,j}]$. We have denoted the event $[\bar{\mathrm{B}}^r_{k,j}|\bar{\mathrm{B}}^r_{k,1},\dots,\bar{\mathrm{B}}^r_{j-1}$, $\bar{\mathrm{A}}^r_1,\dots,\bar{\mathrm{A}}^r_{k-1}]$ by the shorter notation $\mathrm{E}^r_{k,j}$ which is in turn same as the event $\mathrm{D}^r_{k,j}$. Hence in the step (3.11) we have used the inequality (3.9). For all $i < r$, $\mathbf{Pr}[\bar{\mathrm{C}}^r_i] = 1$. Therefore for $k < r$ the quantity $(1-\frac{1}{N}+\frac{\mathbf{Pr}[\bar{\mathrm{C}}^{r-1}_{k-1}]}{N})$ has value 1. So, we can derive step (3.13) from (3.12) □

Here we make a few observations about the probability $\mathbf{Pr}[\bar{\mathrm{C}}^r_i]$. As a corollary of lemma 1 we have,

Corollary 1. *For* $r \ge 3$, $\mathbf{Pr}[\bar{\mathrm{C}}^r_r] \le (1-\frac{1}{N}+\frac{\mathbf{Pr}[\bar{\mathrm{C}}^{r-1}_{r-1}]}{N})$ □

Since, the lemma 1 and corollary 1 are valid for $r \ge 3$ we have to calculate these probabilities explicitly for $r = 2$. Since, according to our assumption all the points in a chain are distinct and the chain length is t, we have:

$$\begin{aligned}
\mathbf{Pr}[\bar{\mathrm{C}}^2_2] &= (1-\frac{t}{N})^t \\
&= 1-\mathrm{O}(\frac{t^2}{N}) && (3.14)
\end{aligned}$$

Similarly, we have,

$$\begin{aligned}\mathbf{Pr}[\bar{\mathrm{C}}_i^2] &= (1-\frac{t}{N})^t \cdot (1-\frac{2t}{N})^t \cdots (1-\frac{(i-1)t}{N})^t \\ &= (1-(1+2+\cdots+i-1)\cdot\frac{t}{N}+\dots)^t \\ &= 1-\mathrm{O}(\frac{i^2t^2}{N}) \end{aligned} \tag{3.15}$$

Next, we prove an important lemma,

Lemma 2. $\mathbf{Pr}[\bar{\mathrm{C}}_i^r] = 1-\mathrm{O}(\frac{i^rt^r}{N^{r-1}})$, *for* $r \geq 2$ *and* $i \geq r$.

Proof: We apply induction on both r and i. The base cases for $r = 2$ and $i = 2$ have already been proved. Here we show the induction step which follows directly from the recurrence relation derived in lemma 1.

$$\begin{aligned}\mathbf{Pr}[\bar{\mathrm{C}}_i^r] &\leq \prod_{k=r}^{i}(1-\frac{1}{N}+\frac{\mathbf{Pr}[\bar{\mathrm{C}}_{k-1}^{r-1}]}{N})^t \\ &= \prod_{k=r}^{i}(1-\mathrm{O}(\frac{(k-1)^{r-1}t^{r-1}}{N^{r-1}}))^t \qquad (3.16)\\ &= (1-\mathrm{O}((r-1)^{r-1}+r^{r-1}+\cdots+(i-1)^{r-1})\cdot\frac{t^{r-1}}{N^{r-1}}+\cdots)^t \\ &= 1-\mathrm{O}(\frac{i^rt^r}{N^{r-1}}) \qquad (3.17)\end{aligned}$$

We replace $\mathbf{Pr}[\bar{\mathrm{C}}_{k-1}^{r-1}]$ by $(1-\mathrm{O}(\frac{(k-1)^{r-1}t^{r-1}}{N^{r-2}}))$ following the induction hypothesis and simplify in step (3.16). □

Next, we state and prove the main result of this analysis:

Theorem 2. $\mathbf{E}(\mathrm{X}_r) = \mathrm{O}(\frac{m^rt^r}{N^{r-1}})$, *for* $\forall r \geq 3$

Proof: From equation (3.7) and inequality (3.9), we get

$$\begin{aligned}\mathbf{Pr}[\wedge_{j=1}^{t}\bar{\mathrm{B}}_{i,j}^r] &\leq \prod_{j=1}^{t}(1-\frac{1}{N}+\frac{\mathbf{Pr}[\bar{\mathrm{C}}_{i-1}^{r-1}]}{N}) \\ &= (1-\frac{1}{N}+\frac{\mathbf{Pr}[\bar{\mathrm{C}}_{i-1}^{r-1}]}{N})^t \\ &= (1-\mathrm{O}(\frac{(i-1)^{r-1}t^{r-1}}{N^{r-1}}))^t \qquad (3.18)\\ &= 1-\mathrm{O}(\frac{(i-1)^{r-1}t^r}{N^{r-1}}) \qquad (3.19)\end{aligned}$$

In step (3.18) we have used lemma 2 to replace $\mathbf{Pr}[\bar{\mathrm{C}}_{i-1}^{r-1}]$ by $(1-\mathrm{O}(\frac{(i-1)^{r-1}t^{r-1}}{N^{r-2}}))$. Now, from equation (3.6) and (3.19), we get,

$$\begin{aligned}\mathbf{Pr}[\mathrm{A}_i^r] &= 1-\mathbf{Pr}[\wedge_{j=1}^{t}\bar{\mathrm{B}}_{i,j}^r] \\ &= \mathrm{O}(\frac{(i-1)^{r-1}t^r}{N^{r-1}})\end{aligned} \tag{3.20}$$

So, from equation (3.5), we have,

$$\begin{aligned}\mathbf{E}(\mathrm{X}_r) &= \sum_{i=1}^{m} \mathbf{Pr}[\mathrm{A}_i^r] \\ &= \sum_{i=1}^{m} \mathrm{O}(\frac{(i-1)^{r-1}t^r}{N^{r-1}}) \\ &= \mathrm{O}(\frac{m^r t^r}{N^{r-1}}) \qquad (3.21)\end{aligned}$$

□

As a corollary of theorem 2, we have,

Corollary 2. *If we use $m = N^{\frac{1}{3}}$ chains with random starting points in distinguished point method with average chain length $N^{\frac{1}{3}}$, then the expected number of three-collisions is constant.*

Proof: Setting $m = N^{\frac{1}{3}}$, $t = N^{\frac{1}{3}}$ and $r = 3$ in equation (3.21), we have,

$$\begin{aligned}\mathbf{E}(\mathrm{X}_3) &= \mathrm{O}(\frac{(N^{\frac{1}{3}})^3(N^{\frac{1}{3}})^3}{N^2}) \\ &= \mathrm{O}(1) \qquad (3.22) \\ & \qquad (3.23)\end{aligned}$$

□

Corollary 2 justifies our choice of $N^{\frac{1}{3}}$ processors which computes chains of length $N^{\frac{1}{3}}$ to find three-collision in our parallel algorithm.

3.3 Time/Processor Tradeoff for Multi-collision

In this section we give a time/processor tradeoff for finding r-collisions based on theorem 2 and describe how our mesh-based parallel algorithm can be used to find r-collisions efficiently for $r > 3$.

Suppose, we are using $m = N^{\alpha}$ processors to find an r-collision and we have chosen our distinguished points in a way such that the average chain length has become $t = N^{\beta}$. Clearly, to make the expected number of r-collisions among the chains constant, according to theorem 2, α and β should satisfy the following tradeoff equation.

$$\alpha + \beta = \frac{r-1}{r} \qquad (3.24)$$

Where, $\alpha > 0$ and $\beta \geq 0$.

Now suppose, in our mesh based parallel algorithm for finding r-collision, we have arranged N^{α} processors in an array of size $N^{\frac{\alpha}{2}} \times N^{\frac{\alpha}{2}}$. The average chain length has been fixed to N^{β} according to the tradeoff equation (3.24). So the average chain computation time is N^{β} and the Schimmler's sorting of the N^{α} outputs takes time $N^{\frac{\alpha}{2}}$. But we want the chain computation time to dominate the overall running time instead of the sorting time. Therefore, we have the following constraint:

$$\frac{\alpha}{2} \leq \beta \qquad (3.25)$$

Solving equation (3.24) and (3.25), we have, $\alpha \leq \frac{2}{3} \cdot \frac{r-1}{r}$ and $\beta \geq \frac{1}{3} \cdot \frac{r-1}{r}$. Therefore our new algorithm can work with less number of processors than the parallel algorithm given in [5]. Our algorithm can find an r-collision with $\frac{2}{3} \cdot \frac{r-1}{r}$ processors running for time $\frac{1}{3} \cdot \frac{r-1}{r}$. So, our algorithm can be used in situations where only limited number of processors are available.

3.4 Conclusion

As outlined in the beginning, we have discussed the problem of finding multi-collisions in random functions in details and have shown different approaches towards this problem. We have been able to provide an efficient algorithm for finding r-collisions for values of $r \geq 3$. We have also succeeded in improving the existing parallelization of the algorithm. Moreover, we we have given a theoretical foundation to the assumptions employed in the previous algorithms. But still there are many interesting and unanswered questions. Consider the most important one – we have mentioned in section 2.3.4 that we can prove a lower bound on the number of map evaluations (i.e. the sequential time complexity) necessary to find an r-collision in a random function. Can we prove such a lower bound for sequential space complexity also? Answer to this question will naturally be of very much importance since we will then know whether we already have the most efficient algorithm for the task or we should keep on trying to improve the space complexity. Same type of questions also arise in the setting of parallel algorithms also. We can ask, whether we can get improved time/processor tradeoff and better overall performance by using some other architecture. In that case which architecture will give the optimum result? We will conclude this thesis by saying that although we know a few interesting things about the problem of finding multi-collisions in random functions, there are still many important questions to be answered and settling these questions will require further intensive research in this subject.

Bibliography

[1] Sourav Mukhopadhyay – *PhD thesis at Indian Statistical Institute*

[2] Amos Fiat, Moni Naor – *Rigorous Time/Memory Tradeoffs for Inverting Functions*, SIAM Journal on Computing, 29(3): pp. 790-803, 1999

[3] Sourav Mukhopadhayay, Palash Sarkar – *A New Cryptanalytic Time/Memory/Data Trade-off Algorithm*, eprint archive on cryptology.

[4] Elad Barkan, Eli Biham and Adi Shamir – *Rigorous Bounds on Cryptanalytic Time/Memory Trade-offs*, CRYPTO 2006, LNCS 4117, pp 1-21, 2006

[5] Antoine Joux and Stefan Lucks – *Improved Generic Algorithms for 3-Collisions*, eprint cryptology archive

[6] Gabriel Nivasch – *Cycle Detection Using a Stack*, Information Processing Letters 90(2004), pp 135-140, 2004

[7] Philippe Flajolet, Andrew M. Odlyzko – *Random Mapping Statistics*

[8] Daniel J. Bernstein – *Understanding Brute Force*